Introducing R

Online Version at
https://grodri.github.io/R

Germán Rodríguez
Princeton University

Updated December 2022

1 Introduction

R is a powerful environment for statistical computing that runs on several platforms. These notes are written specially for users running the Windows version, but most of the material applies to the macOS and Linux versions as well. Following some bibliographic remarks and tips for getting started, I describe reading and examining data, and fitting linear and generalized linear models. I close with a few pointers and references on where to go from here. I have tried to introduce key features of R as needed by students in my statistics classes. As a result, I often postpone (or altogether omit) discussion of some of the more powerful features of R as a programming language.

1.1 The R Language and Environment

R was first written as a research project by Ross Ihaka and Robert Gentleman, and is now under active development by a group of statisticians called "the R core team", with a home page at https://www.r-project.org.

R was designed to be "not unlike" the S language developed by John Chambers and others at Bell Labs. A commercial version of S with additional features was developed and marketed as S-Plus by Statistical Sciences, eventually becoming part of TIBCO Spotfire. You can view R and S-Plus as alternative implementations of the same underlying S language. The modern R implementation, however, is by far the most popular.

R is available free of charge and is distributed under the terms of the Free Software Foundation's GNU General Public License. You can download the program from the Comprehensive R Archive Network (CRAN). Ready-to-run binaries are available for Linux, macOS, and Windows. The source code is also available for download and can be compiled for other platforms.

1.2 Bibliographic Remarks

S was first introduced by Becker and Chambers (1984) in what's known as the "brown" book. The new S language was described by Becker, Chambers and Wilks (1988) in the "blue"

book. Chambers and Hastie (1992) edited a book discussing statistical modeling in S, called the "white" book. The latest version of the S language is described by Chambers (1998) in the "green" book, but R is largely an implementation of the versions documented in the blue and white books. Chamber's latest books, Chambers (2008) and Chambers (2016), focus on programming with R.

Venables and Ripley (2002) have written an excellent book on Modern Applied Statistics with S-PLUS that is now in its fourth edition. The latest edition is particularly useful to R users because the main text explains differences between S-Plus and R where relevant. A companion volume called *S Programming* appeared in 2000 and applies to both S-Plus and R (Venables and Ripley 2000). These authors have also made available in their website an extensive collection of complements to their books, follow the links at MASS 4.

There is now an extensive and rapidly growing literature on R. Good introductions include the books by Krause and Olson (1997), Dalgaard (2008), and Braun and Murdoch (2016). Beginners will probably benefit from working through the examples in Hothorn and Everitt (2014) *A Handbook of Statistical Analyses Using R*, now in its third edition, or Fox (2002)'s companion to applied regression. Among more specialized books my favorites include Murrell (2006), an essential reference on R graphics, Pinheiro and Bates (2000), a book on mixed models, and Therneau and Grambsch (2000)'s book on survival models. (Therneau wrote the survival analysis code used in S-Plus and R.)

Hadley Wickham has made a number of ground-breaking contributions to R that deserve special mention. He is the author of the `ggplot2` package (Wickham 2016), a very popular graphics package that has brought to R the grammar of graphics proposed by Wilkinson (2005). He has also contributed a number of data-management packages, of which the most notable is `dplyr`, and has advocated the use of "tidy" datasets. His approach to data management is explained in detail in Wickham and Grolemund (2017), with its own website at https://www.tidyverse.org/. He has also written an advanced book on programming R, now in its second edition (Wickham 2019). You will find most of his work available online; follow the links in the list of references at the end of this tutorial.

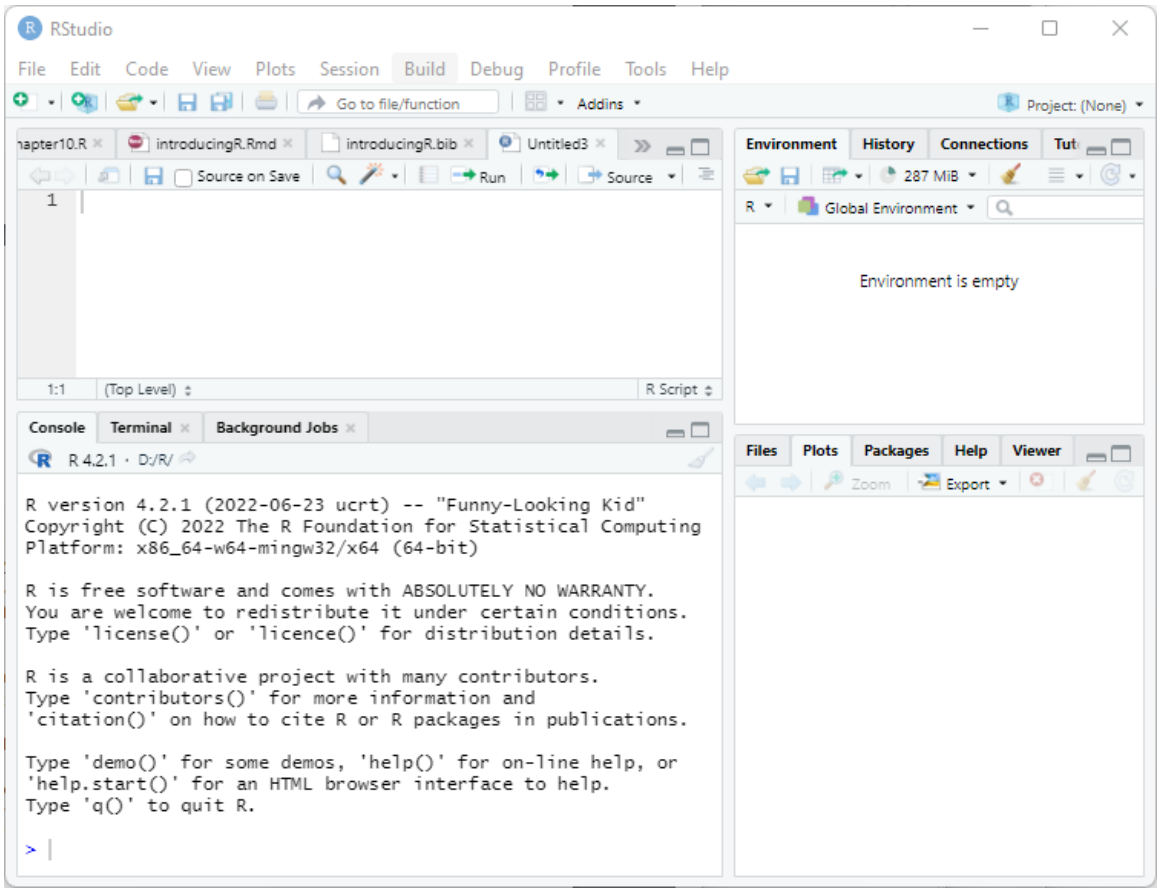
The official R manuals are distributed as PDF files. These include *An Introduction to R* (a nice 100-page introduction), a manual on *R Data Import/Export* describing facilities for transferring data to and from other packages, and useful notes on *R installation and Administration*. More specialized documents include a draft of the *R Language Definition*, a guide to *Writing R Extensions*, documentation on *R Internals* including coding standards, and finally the massive *R Reference Index* (~3000 pages). The online help facility is excellent.

2 Getting Started

Obviously the first thing you need to do is download a copy of R. This tutorial was updated using version 4.2.1, released in June 2022. New releases occur every six months or so. You will find the latest version and binaries for Windows, macOS, and Linux in the Comprehensive R Archive Network. Find the mirror nearest you and follow the links. The Windows installer is fairly easy to use and, after agreeing to the license terms, lets you choose which components

you want to install. Additional packages can always be installed directly from R at a later time.

The next thing you should do is install R Studio, a nice Integrated Development Environment (IDE) for R, that is also available for Linux, macOS and Windows. You can download the free open source edition of R Studio Desktop from https://rstudio.com/products/rstudio/. Once the software is installed, it will find and integrate your R installation.



The panel on the lower-left corner of the IDE shows the console, which is described below. Graphs and help will appear on the bottom right panel. The panel on the top left can be used to type an R script to be run later.

If you do not install R Studio, you can still run the R graphical user interface (GUI). I recommend that you create a shortcut on the desktop. When R starts you see a console. Graphs appear on a separate window.

2.1 The Console

The console in R Studio (or R) is where you type R expressions and see text results. You are prompted to type some input with the greater than symbol. To quit R Studio (and R) type

```
> q()
```

Note the parentheses after the `q`. This is because in R you do not type commands, but rather call *functions* to achieve results, even quit! To call a function you type the name followed by the arguments in parentheses. If the function takes no arguments, you just type the name followed by left and right parenthesis. (If you forget the parentheses and type just the name of the function, R will list the source code.) After typing `q()` you are asked whether you want to save the current work environment. You can skip this prompt by typing `q("no")`.

You should also know about the `help()` function, which uses the window on the bottom right (opens a separate help window in R). The function can be called with arguments to obtain help about specific features of R, for example `help(plot)`. A shortcut for help on a topic is a question mark followed by the topic, as in `?plot`.

The console allows input editing. You will find that the left and right arrow keys, home, end, backspace, insert, and delete work exactly as you would expect. You also get an input history: the up and down arrow keys can be used to scroll through recent input lines. Thus, if you make a mistake, all you need to do is press the up key to recall your last expression and edit it.

You can also type a script on the top left window of R Studio and then click on "Run".

2.2 Expressions and Assignments

R works like a calculator, you type an expression and get the answer:

```
> 1 + 2
```

```
[1] 3
```

The standard arithmetic operators are `+`, `-`, `*`, and `/` for add, subtract, multiply and divide, and `^` for exponentiation, so `2^3 = 8`. These operators have the standard precedence, with exponentiation highest and addition/subtraction lowest, but you can always control the order of evaluation with parentheses. You can use mathematical functions, such as `sqrt()`, `exp()`, and `log()`. For example

```
> log(0.3/(1 - 0.3))
```

```
[1] -0.8472979
```

R also understands the relational operators `<=`, `<`, `==`, `>`, `>=` and `!=` for less than or equal, less than, equal, greater than, greater than or equal, and not equal. These can be used to create logical expressions that take values `TRUE` and `FALSE`. (Or `T` and `F` for short, but these abbreviations are not recommended because it is possible to assign other values to them.) Logical expressions may be combined with the logical operators `|` for OR and `&` for AND, as shown further below.

The results of a calculation may be assigned to a named object. The assignment operator in R is `<-`, read as "gets", but by popular demand R now accepts the equal sign as well, so `x <- 2` and `x = 2` both assign the value 2 to a variable (technically an object) named `x`. I

prefer using `<-`, but you can use either. Note that `=` is also used for named arguments of functions.

Typing a name prints its contents. The name `pi` is used for the constant π. Thus,

```
> s <- pi/sqrt(3)
> s
```

```
[1] 1.813799
```

assigns $\pi/\sqrt{3}$ to the variable `s` and prints its value.

Names may contain letters, numbers or periods, and (starting with 1.9.0) the underscore character, but must start with a letter or period. (I recommend you always start names with a letter.) Thus, `v.one` and `v_one` are valid names, but `v one` is not (because it includes a space).

Warning: R is case sensitive, `v.one`, `V.one` and `v.One` are all different names.

R objects exist during your session but vanish when you exit. However, as noted earlier you will be asked if you want to save an image of your workspace before you leave. You can also save individual objects to disk, see `help(save)`.

Note that assignments are expressions too; you can type x <- y <- 2 and both x and y will get 2. This works because the assignment y <- 2 is also an expression that takes the value 2.

Exercise: What's the difference between `x == 2` and `x = 2`? Use the console to find out.

2.3 Vectors and Matrices

So far we have worked with scalars (single numbers) but R is designed to work with **vectors** as well. The function `c()`, which is short for catenate (or concatenate if you prefer) can be used to create vectors from scalars or other vectors

```
> x <- c(1, 3, 5, 7)
> x
```

```
[1] 1 3 5 7
```

The colon operator `:` can be used to generate a sequence of numbers

```
> x <- 1:10
> x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

You can also use the `seq()` function to create a sequence given the starting and stopping points and an increment. For example here are eleven values between 0 and 1 in steps of 0.1:

```
> seq(0, 1, 0.1)
```

```
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

Another function that is useful in creating vectors is `rep()` for repeat or replicate. For example `rep(3, 4)` replicates the number three four times. The first argument can be a vector, so `rep(x, 3)` replicates the entire vector `x` three times. If both arguments are vectors of the same size, then each element of the first vector is replicated the number of times indicated by the corresponding element in the second vector. Consider this example:

```
> rep(1:3, 2)
```

```
[1] 1 2 3 1 2 3
```

```
> rep(1:3, c(2, 2, 2))
```

```
[1] 1 1 2 2 3 3
```

The first call repeats the vector `1:3` twice. The second call repeats each element of `1:3` twice, and could have been written `rep(1:3, rep(2, 3))`, a common R idiom.

R operations are vectorized. If `x` is a vector, then `log(x)` is a vector with the logs of the elements of `x`. Arithmetic and relational operators also work element by element. If `x` and `y` are vectors of the same length, then `x + y` is a vector with elements equal to the sum of the corresponding elements of `x` and `y`. If `y` is a scalar, it is added to each element of `x`. If `x` and `y` are vectors of different lengths, the shorter one is recycled as needed, perhaps a fractional number of times (in which case you get a warning).

The logical operators `|` for *or* and `&` for *and* also work element by element. (The double operators `||` for *or* and `&&` for *and* work only on the first element of each vector, and use shortcut evaluation; they are used mostly in writing R functions, and will not be discussed further.)

```
> a <- c(TRUE, TRUE, FALSE, FALSE)
> b <- c(TRUE, FALSE, TRUE, FALSE)
> a & b
```

```
[1]  TRUE FALSE FALSE FALSE
```

The number of elements of a vector is returned by the function `length()`. Individual elements are addressed using subscripts in square brackets, so `x[1]` is the first element of `x`, `x[2]` is the second, and `x[length(x)]` is the last.

The subscript can be a vector itself, so `x[1:3]` is a vector consisting of the first three elements of `x`. A negative subscript *excludes* the corresponding element, so `x[-1]` returns a vector with all elements of `x` except the first one.

Interestingly, a subscript can also be a logical expression, in which case you get the elements for which the expression is `TRUE`. For example, to list the elements of `x` that are less than `5` we write

```
> x[x < 5]
```

```
[1] 1 2 3 4
```

I read this expression "x such that x is less than 5". That works because the subscript `x < 5` is this vector

```
> x < 5
```

```
 [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

R's subscripting facility is extremely powerful. You may find that it takes a while to get used to it, but eventually the language becomes natural.

R also understands **matrices** and higher dimensional arrays. The following function creates a 3 by 4 matrix and fills it by columns with the numbers 1 to 12:

```
> M <- matrix(1:12, 3, 4)
> M
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

The elements of a matrix may be addressed using the row and column subscripts in square brackets, separated by a comma. Thus, `M[1, 1]` is the first element of `M` (in row 1 and column 1).

A subscript may be left blank to select an entire row or column: `M[1,]` is the first row, and `M[,1]` is the first column of `M`. Any of the subscripts may be a vector, so `M[1:2, 1:2]` is the upper-left 2 by 2 corner of `M`. Try it.

The number of rows and columns of a matrix are returned by the functions `nrow()` and `ncol()`. To transpose a matrix use the function `t()`. The matrix multiplication operator is `%*%`. Matrix inversion is done by the function `solve()`. See the linear regression section for an example.

Exercise: How do you list the last element of a matrix?

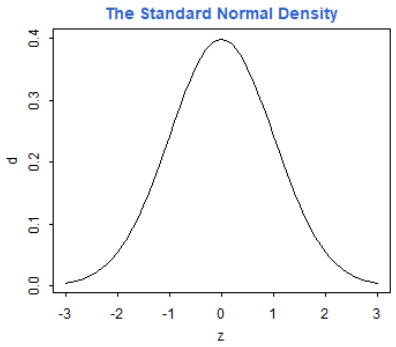
2.4 Simple Graphs

R has very extensive and powerful graphic facilities. In the example below we use `seq()` to create equally spaced points between -3 and 3 in steps of 0.1 (that's 61 points). Then we call the function `dnorm()` to calculate the standard normal density evaluated at those points, we plot it, and add a title in a nice shade of blue. Note that we are able to add the title to the current plot in a separate call.

```
> z <- seq(-3, 3, 0.1)
> d <- dnorm(z)
```

```
> plot(z, d, type="l")
> title("The Standard Normal Density", col.main="#3366CC")
```

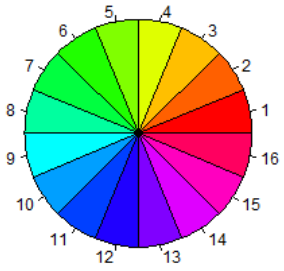
Arguments to a function can be specified by position or by name. The `plot()` function expects the first two arguments to be vectors giving the `x` and `y` coordinates of the points to be plotted. We also specified the `type` of plot. Since this is one of many optional parameters (type `?plot` for details), we specified it by name as `type="l"` (the letter el). This indicates that we want the points joined to form a line, rather than the default which is to plot discrete points. Note that R uses the equal sign to specify named arguments to a function.



The `title()` function expects a character string with the title as the first argument. We also specified the optional argument `col.main="#3366CC"` to set the color of the title. Here I specified the red, green and blue components of the color in hexadecimal notation; this particular choice matches the headings on my website. There are also 657 named colors to choose from, type `colors()` to see their names.

The next example is based on a demo included in the R distribution, and is simply meant to show off R's use of colors. We use the `pie()` function to create a chart with 16 slices. The slices are all the same width, but we fill them with different colors obtained using the `rainbow()` function.

```
> pie(rep(1, 16), col = rainbow(16))
```



Note the use of the rep function to replicate the number one 16 times. To see how one can specify colors and labels for the slices, try calling pie with arguments `1:4`, `c("r", "g", "b","w")` and `col = c("red,"green","blue","white")`.

To save a graph in R Studio click on the "Export" button on the graph window. (In R make sure the focus is on the graph window and choose `File | Save as`, from the menu.) You can save as an image or a PDF file.

One of the image formats is `png`, which makes it easy to include R graphs in web pages. R also supports `jpeg`, but I think `png` is better than `jpeg` for statistical plots. All graphs on the web version of this tutorial are in `png` format.

Alternatively, you can copy the graph to the clipboard by choosing "Clipboard". (In R select `File | Copy to clipboard`.) You get a choice of two formats: bitmap and metafile. I recommend that you use the metafile format because it's more flexible. You can then paste the graph into a word processing or spreadsheet document. (In R you can also print the graph using `File | Print`.)

Exercise: Simulate 20 observations from the regression model $Y = \alpha + \beta x + e$ using the `x` vector generated above. Set $\alpha = 1$ and $\beta = 2$. Use standard normal errors generated as `rnorm(20)`, where 20 is the number of observations.

3 Reading and Examining Data

R can handle several types of data, including numbers, character strings, vectors and matrices, as well as more complex data structures. In this section I describe **data frames**, the preferred way to organize data for statistical analysis, explain how to read data from an external file into a data frame, and show how to examine the data using simple descriptive statistics and informative plots.

3.1 Lists and Data Frames

An important data structure that we have not discussed so far is the **list**. A list is a set of objects that are usually named and can be anything: numbers, character strings, matrices or even lists.

Unlike a vector, whose elements must all be of the same type (all numeric, or all character), the elements of a list may have different types. Here's a list with two components created using the function `list()`:

```
> person <- list(name="Jane", age=24)
```

Typing the name of the list prints all elements.

```
> person
```

```
$name
[1] "Jane"

$age
[1] 24
```

You can extract a component of a list using the extract operator `$`. For example we can list just the `name` or `age` of this person:

```
> person$name
```

```
[1] "Jane"
```

```
> person$age
```

```
[1] 24
```

Individual elements of a list can also be accessed using their indices or their names as subscripts. For example we can get the name using `person[1]` or `person["name"]`. You can use single or double square brackets. If you use single brackets, as we did here, you get a list with the name. If you use double brackets you get just the name. Try `person[[1]]` or `person[["name"]]` to see the difference.

A **data frame** is essentially a rectangular array containing the values of one or more variables for a set of units. The frame also contains the names of the variables, the names of the observations, and information about the nature of the variables, including whether they are numerical or categorical.

Internally, a data frame is a special kind of list, where each element is a vector of observations on a variable. Data frames look like matrices, but can have columns of different types. This makes them ideally suited for representing datasets, where some variables can be numeric and others can be categorical.

Data frames (like matrices) can also accommodate missing values, which are coded using the special symbol `NA`. Most statistical procedures, however, omit all missing values.

Data frames can be created from vectors, matrices or lists using the function `data.frame()`, but more often than not one will read data from an external file, as shown in the next two sections.

3.2 Free-Format Input

Free-format data are text files containing numbers or character strings separated by spaces. Optionally the file may have a header containing variable names. Here's an excerpt of a data file containing information on three variables for 20 countries in Latin America:

```
              setting  effort   change
Bolivia            46       0        1
Brazil             74       0       10
Chile              89      16       29
  ... lines omitted ...
Venezuela          91       7       11
```

This small dataset includes an index of social setting, an index of family planning effort, and the percent decline in the crude birth rate between 1965 and 1975. The data are available on my website in a file called `effort.dat` that includes a header with the variable names.

R can read the data directly from the web:

```
> fpe <- read.table("https://grodri.github.io/datasets/effort.dat")
```

The function used to read data frames is `read.table()`. The argument is a character string giving the name of the file containing the data, but here we have given it a fully qualified url (uniform resource locator), and that's all it takes.

Alternatively, you could download the data and save them in a local file, or just cut and paste the data from the browser to an editor, and then save them. Make sure the file ends up in R's working directory, which you can find out by typing `getwd()`. If that is not the case, you can use a fully qualified path name or change R's working directory by calling `setwd()` with a string argument. Remember to double up your backward slashes (or use forward slashes instead) when specifying paths in Windows.

Here we assigned the data to an object called `fpe`. To print the object you simply type its name

```
> fpe
```

```
               setting effort change
Bolivia             46      0      1
Brazil              74      0     10
Chile               89     16     29
Colombia            77     16     25
CostaRica           84     21     29
Cuba                89     15     40
DominicanRep        68     14     21
Ecuador             70      6      0
ElSalvador          60     13     13
Guatemala           55      9      4
Haiti               35      3      0
Honduras            51      7      7
Jamaica             87     23     21
Mexico              83      4      9
Nicaragua           68      0      7
Panama              84     19     22
Paraguay            74      3      6
Peru                73      0      2
TrinidadTobago      84     15     29
Venezuela           91      7     11
```

In this example R detected correctly that the first line in our file was a header with the variable names. It also inferred correctly that the first column had the observation names. (Well, it did so with a little help; I made sure the row names did not have embedded spaces, hence `CostaRica`. Alternatively, I could have used `"Costa Rica"` in quotes as a row name.)

You can always tell R explicitly whether or not you have a header by specifying the optional argument `header=TRUE` or `header=FALSE` to the `read.table()` function. This is important if you have a header but lack row names, because R's guess is based on the fact that the header line has one fewer entry than the next row, as it did in our example.

If your file does not have a header line, R will use the default variable names `V1`, `V2`, ..., etc. To override this default use `read.table()`'s optional argument `col.names` to assign variable names. This argument takes a vector of names. So, if our file did *not* have a header we could have used the command

```
> fpe <- read.table("noheader.dat", col.names=c("setting","effort","change"))
```

Don't worry if this command doesn't fit in a line. R code can be continued automatically in a new line simply by making it obvious that we are not done, for example ending the line with a comma, or having an unclosed left parenthesis. R responds by prompting for more with the continuation symbol `+` instead of the usual prompt `>`.

If your file does not have observation names, R will simply number the observations from 1 to n. You can specify row names using `read.table()`'s optional argument `row.names`, which works just like `col.names`; type `?data.frame` for more information. (I should mention that in a "tidy" world row names should just be another column, but classic R treats them as observation indices.)

There are two closely related functions that can be used to get or set variable *and* observation names at a later time. These are called `names()`, for the variable or column names, and `row.names()` for the observation or row names. Thus, if our file did not have a header we could have read the data and then changed the default variable names using the `names()` function:

```
> fpe <- read.table("noheader.dat")
> names(fpe) <- c("setting","effort","change")
```

Technical Note: If you have a background in other programming languages, you may be surprised to see a function call on the left hand side of an assignment. These are special *replacement* functions in R. They extract an element of an object and then replace its value.

In our example all three-variables were numeric. R will handle categorical variables with no problem, including factors and string variables. In Section 4 we will create a factor, basically a categorical variable that takes one of a finite set of values called *levels*, by grouping a numeric covariate into categories. In Section 5 we will read a dataset that includes string variables with values such as "low" and "high". These can be converted to factors or kept as character data.

Exercise: Use a text editor to create a small file with the following three lines:

```
a b c
1 2 3
4 5 6
```

Read this file into R so the variable names are `a`, `b` and `c`. Now delete the first row in the file, save it, and read it again into R so the variable names are still `a`, `b` and `c`.

3.3 Fixed-Format Input

Suppose the family planning effort data had been stored in a file containing only the actual data (no country names or variable names) in a fixed format, with social setting in character positions (often called columns) 1-2, family planning effort in positions 3-4 and fertility change in positions 5-6. This is a fairly common way to organize large datasets.

The following call will read the data into a data frame and name the variables:

```
> fpe <- read.table("fixedformat.dat",  col.names = c("setting", "effort",
"change"), sep=c(1, 3, 5))
```

Here I assume that the file in question is called `fixedformat.dat`. I assign column names just as before, using the `col.names` parameter. The novelty lies in the next argument, called `sep`, which is used to indicate how the variables are separated. The default is white space,

which is appropriate when the variables are separated by one or more blanks or tabs. If the data are separated by commas, a common format with spreadsheets, you can specify `sep = ","`. Here I created a vector with the numbers 1, 3 and 5 to specify the character position (or column) where each variable starts. Type `?read.table` for more details.

3.4 Printing Data and Summaries

You can refer to any variable in the `fpe` data frame using the extract operator `$`. For example to look at the values of the fertility change variable, type

```
> fpe$change
```

```
 [1]  1 10 29 25 29 40 21  0 13  4  0  7 21  9  7 22  6  2 29 11
```

and R will list a vector with the values of change for the 20 countries. You can also define `fpe` as your default dataset by "attaching" it to your session:

```
> attach(fpe)
```

If you now type the name `effort` by itself, R will now look for it in the `fpe` data frame. If you are done with a data frame, you can detach it using `detach(fpe)`. While `attach()` can save typing, experience has shown that it can also lead to problems, suggesting it is best avoided. For example, if you already have an object named `effort`, that will mask the object in `fpe`. My advice is to always specify the data frame name, as we do below.

To obtain simple descriptive statistics on these variables try the `summary()` function:

```
> summary(fpe)
```

```
    setting         effort          change
 Min.   :35.0   Min.   : 0.00   Min.   : 0.00
 1st Qu.:66.0   1st Qu.: 3.00   1st Qu.: 5.50
 Median :74.0   Median : 8.00   Median :10.50
 Mean   :72.1   Mean   : 9.55   Mean   :14.30
 3rd Qu.:84.0   3rd Qu.:15.25   3rd Qu.:22.75
 Max.   :91.0   Max.   :23.00   Max.   :40.00
```

As you can see, we get the min and max, 1st and 3rd quartiles, median and mean. For categorical variables you get a table of counts. Alternatively, you may ask for a summary of a specific variable. Or use the functions `mean()` and `var()` for the mean and variance of a variable, or `cor()` for the correlation between two variables, as shown below:

```
> mean(fpe$effort)
```

```
[1] 9.55
```

```
> cor(fpe$effort, fpe$change)
```

```
[1] 0.8008299
```

Elements of data frames can be addressed using the subscript notation introduced in Section 2.3 for vectors and matrices. For example to list the countries that had a family planning effort score of zero we can use

```
> fpe[fpe$effort == 0, ]
```

```
          setting effort change
Bolivia        46      0      1
Brazil         74      0     10
Nicaragua      68      0      7
Peru           73      0      2
```

This works because the expression `fpe$effort == 0` selects the rows (countries) where the effort score is zero, while leaving the column subscript blank selects all columns (variables).

The fact that the rows are named allows yet another way to select elements: by name. Here's how to print the data for Chile:

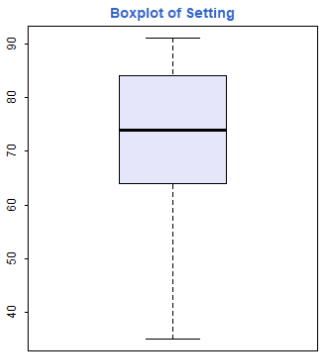
```
> fpe["Chile", ]
```

```
      setting effort change
Chile      89     16     29
```

Exercise: Can you list the countries where social setting is high (say above 80) but effort is low (say below 10)? Hint: recall the element-by-element logical operator `&`.

3.5 Plotting Data

Probably the best way to examine the data is by using graphs. Here's a boxplot of setting. Inspired by a demo included in the R distribution, I used custom colors for the box ("lavender", specified using a name R recognizes) and the title (`#3366CC`).

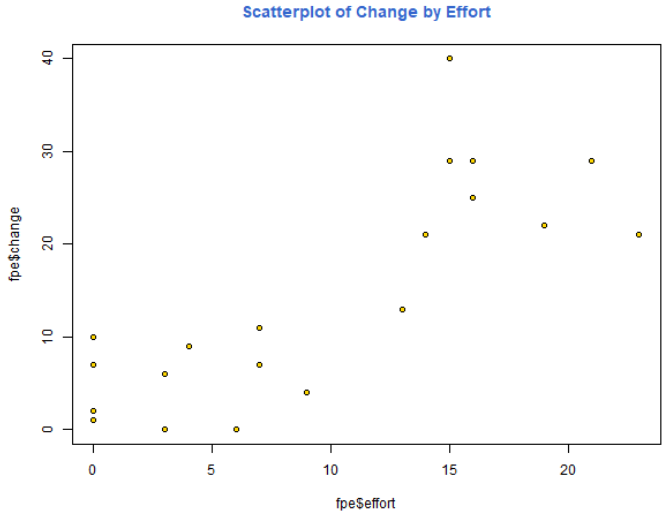


As noted earlier, R can save a plot as a png or jpeg file, so that it can be included directly on a web page. Other formats available are postscript for printing and windows metafile for embedding in other applications. Note also that you can cut and paste a graph to insert it in another document.

```
> boxplot(fpe$setting, col="lavender")
> title("Boxplot of Setting", col.main="#3366CC")
```

Here's a scatterplot of change by effort, so you can see what a correlation of 0.80 looks like:

```
> plot(fpe$effort, fpe$change, pch=21, bg="gold")
> title("Scatterplot of Change by Effort", col.main="#3366CC")
```



I used two optional arguments that work well together: `pch=21` selects a special plotting symbol, in this case a circle, that can be colored and filled; and `bf="gold"` selects the fill color for the symbol. I left the perimeter black, but you can change this color with the `col` argument.

To identify points in a scatterplot use the `identify()` function. Try the following on the graph window:

```
> identify(fpe$effort, fpe$change, row.names(fpe), ps = 9)
```

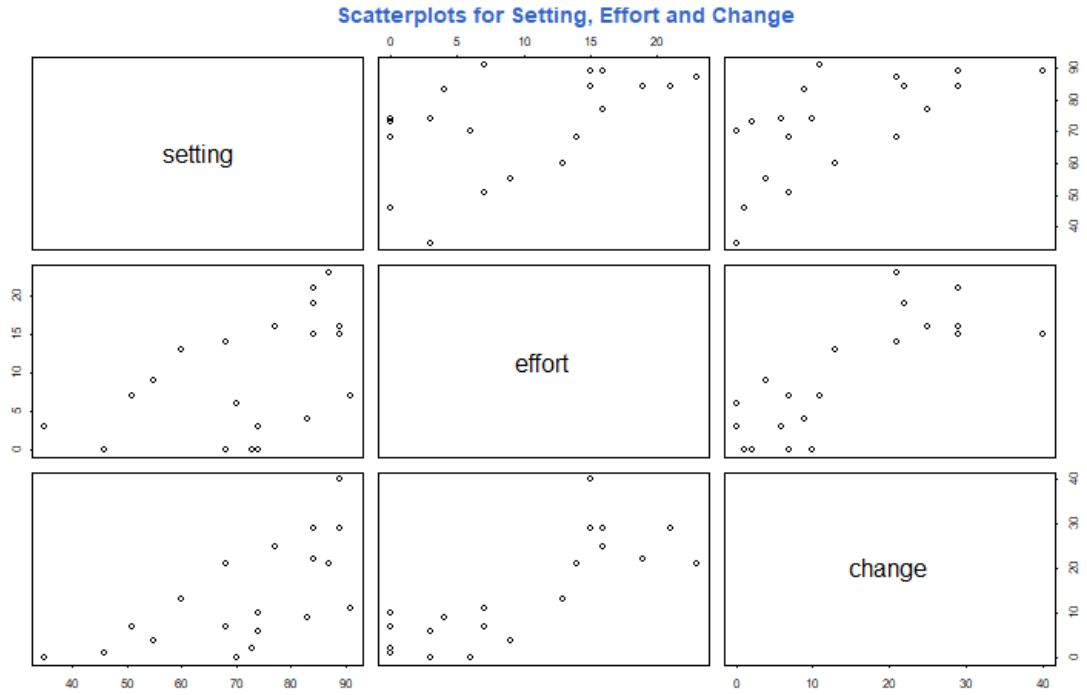
The first three parameters to this function are the x and y coordinates of the points and the character strings to be used in labeling them. The `ps` optional argument specifies the size of the text in points; here I picked 9-point labels.

Now click within a quarter of an inch of the points you want to identify. R Studio will note that "locator is active". When you are done clicking press the `Esc` key. The labels will then appear next to the points you clicked on. (If you are using the R GUI, the labels will appear as you click on the points.)

Which country had the most effort but only moderate change? Which one had the most change?

Another interesting plot to try is `pairs()`, which draws a scatterplot matrix. In our example try

```
pairs(fpe)
title("Scatterplots for Setting, Effort and Change", col.main="#3366CC")
```



The result is a 3 by 3 matrix of scatterplots, with the variable names down the diagonal and plots of each variable against every other one.

Before you quit this session consider saving the `fpe` data frame. To do this use the `save()` function

```
> save(fpe, file="fpe.Rdata")
> load("fpe.Rdata")
```

The first argument specifies the object to be saved, and the `file` argument provides the name of a file, which will be in the working directory unless a full path is given. (Remember to double-up your backslashes in Windows, or use forward slashes instead.)

By default R saves objects using a compact binary format which is portable across all R platforms. There is an optional argument `ascii` that can be set to `TRUE` to save the object as ASCII text. This option was handy to transfer R objects across platforms, but is no longer needed.

You can also save an image of your entire workspace, including all objects you have defined, and then load everything again, using

```
> save.image(file = "workspace.Rdata")
> load("workspace.Rdata")
```

In R Studio you can also do this using the Environment tab on the top right; click on the floppy disk image to save the workspace, or on the folder with an arrow to load a workspace. (In the R Gui you can use the main menu; choose `File|Save` and `File|Load`.)

When you quit R using `q()` you will be prompted to save the workspace, unless you skip this safeguard by typing `q("no")`.

Exercise: Use R to create a scatterplot of change by setting, cut and paste the graph into a document in your favorite word processor, and try resizing and printing it. I recommend that you use the windows metafile format for the cut and paste operation.

4 Linear Models

Let us try some linear models, starting with multiple regression and analysis of covariance models, and then moving on to models using regression splines. In this section I will use the data read in Section 3, so make sure the `fpe` data frame is still available (or read it again).

4.1 Fitting a Model

To fit an ordinary linear model with fertility change as the response and setting and effort as predictors, try

```
> lmfit <- lm( change ~ setting + effort, data = fpe )
```

Note first that `lm()` is a function, and we assign the result to an object that I choose to call `lmfit` (for linear model fit). This stores the results of the fit for later examination.

The first argument to `lm()` is a model formula, which has the response on the left of the tilde `~` (read "is modeled as"), and a Wilkinson-Rogers model specification formula on the right. R uses

`+` to combine elementary terms, as in `A + B`
`:` for interactions, as in `A:B`;
`*` for both main effects and interactions, so `A * B = A + B + A:B`

A nice feature of R is that it lets you create interactions between categorical variables, between categorical and continuous variables, and even between numeric variables (it just creates the cross-product).

We also used the `data` argument to specify the data frame containing these variables.

4.2 Examining a Fit

Let us look at the results of the fit. One thing you can do with `lmfit`, as you can with any R object, is print it.

```
> lmfit


Call:
lm(formula = change ~ setting + effort, data = fpe)

Coefficients:
(Intercept)      setting       effort
```

```
   -14.4511       0.2706       0.9677
```

The output includes the model formula and the coefficients. You can get a bit more detail by using the `summary()` function:

```
> summary(lmfit)


Call:
lm(formula = change ~ setting + effort, data = fpe)

Residuals:
     Min       1Q   Median       3Q      Max
-10.3475  -3.6426   0.6384   3.2250  15.8530

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -14.4511     7.0938  -2.037 0.057516 .
setting       0.2706     0.1079   2.507 0.022629 *
effort        0.9677     0.2250   4.301 0.000484 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 6.389 on 17 degrees of freedom
Multiple R-squared:  0.7381,    Adjusted R-squared:  0.7073
F-statistic: 23.96 on 2 and 17 DF,  p-value: 1.132e-05
```

The output includes a more conventional table with parameter estimates and standard errors, as well the residual standard error and multiple R-squared. (You could also get the matrix of correlations among parameter estimates, by adding the option `correlation = TRUE` in the call to `summary()`, but that is a bit too much detail.)

To get a hierarchical analysis of variance table corresponding to introducing each of the terms in the model one at a time, in the same order as in the model formula, try the `anova()` function:

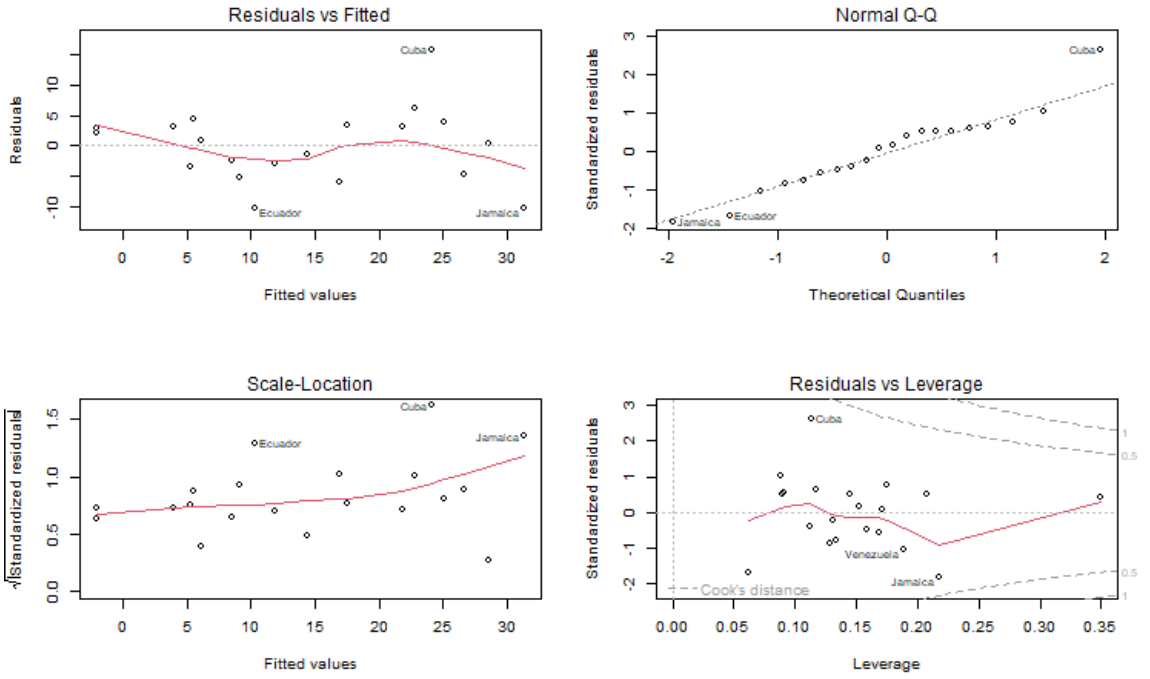
```
> anova(lmfit)

Analysis of Variance Table

Response: change
          Df  Sum Sq Mean Sq F value    Pr(>F)
setting    1 1201.08 1201.08  29.421 4.557e-05 ***
effort     1  755.12  755.12  18.497 0.0004841 ***
Residuals 17  694.01   40.82
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Alternatively, you can plot the results using

```
plot(lmfit)
```

This will produce a set of four plots: residuals versus fitted values, a Q-Q plot of standardized residuals, a scale-location plot (square roots of standardized residuals versus fitted values, and a plot of residuals versus leverage, that adds bands corresponding to Cook's distances of 0.5 and 1.

R Studio (and R) will prompt you to press Enter before showing each graph, but we can do better. Type `par(mfrow = c(2, 2))` to set your graphics window to show four plots at once, in a layout with 2 rows and 2 columns. Then redo te graph using `plot(lmfit)`. To go back to a single graph per window use `par(mfrow = c(1, 1))`. There are many other ways to customize your graphs by setting high-level parameters, type `?par` to learn more.



Technical Note: You may have noticed that we have used the function `plot()` with all kinds of arguments: one or two variables, a data frame, and now a linear model fit. In R jargon, `plot()` is a *generic* function. It checks for the kind of object that you are plotting, and then calls the appropriate (more specialized) function to do the work. There are actually many plot functions in R, including `plot.data.frame()` and `plot.lm()`, and you can let R figure out which one to call.

4.3 Extracting Results

There are some specialized functions that allow you to extract elements from a linear model fit. For example

```
> fitted(lmfit)
```

```
      Bolivia        Brazil         Chile      Colombia     CostaRica
    -2.004026      5.572452     25.114699     21.867637     28.600325
         Cuba  DominicanRep       Ecuador    ElSalvador     Guatemala
    24.146986     17.496913     10.296380     14.364491      9.140694
        Haiti      Honduras       Jamaica        Mexico     Nicaragua
    -2.077359      6.122912     31.347518     11.878604      3.948921
       Panama      Paraguay           Peru TrinidadTobago     Venezuela
    26.664898      8.475593      5.301864     22.794043     16.946453
```

extracts the fitted values. In this case it will also print them, because we did not asign them to anything. (The longer form `fitted.values()` is an alias.)

To extract the coefficients use the `coef()` function (or the longer form `coefficients()`)

```
> coef(lmfit)
```

```
(Intercept)     setting      effort
-14.4510978   0.2705885   0.9677137
```

To get the residuals, use the `resids()` function (or the longer form `residuals()`). There is a `type` argument that lets you choose several types of residuals, type `?residuals.lm` for information. I find more useful the `rstudent()` function that returns standardized residuals:

```
> rstudent(lmfit)
```

```
      Bolivia        Brazil         Chile      Colombia     CostaRica
   0.51666939     0.75316960     0.63588630     0.50233619     0.06666317
        Cuba  DominicanRep       Ecuador    ElSalvador     Guatemala
   3.32236668     0.56318276    -1.76471053    -0.22267614    -0.85483603
       Haiti      Honduras       Jamaica        Mexico     Nicaragua
   0.39308668     0.14477900    -1.98177567    -0.47988042     0.50479726
      Panama      Paraguay           Peru TrinidadTobago     Venezuela
  -0.77508737    -0.40082283    -0.55507263     1.01832414    -1.03565220
```

If you are curious to see exactly what a linear model fit produces, try the function

```
> names(lmfit)
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
```

which lists the named components of a linear fit. All of these objects may be extracted using the `$` operator. However, if there is a special extractor function such as `coef()` or `resid()`, you are encouraged to use it.

4.4 Factors and Covariates

So far our predictors have been continuous variables or *covariates*. We can also use categorical variables or *factors*. Let us group family planning effort into three categories:

```
> fpe$effortg <- cut(fpe$effort, breaks = c(-1, 4, 14, 100),
+   label = c("weak", "moderate", "strong"))
```

The function `cut()` creates a factor or categorical variable. The first argument is an input vector, the second is a vector of breakpoints, and the third is a vector of category labels. Note that there is one more breakpoint than there are categories. All values greater than the i-th breakpoint and less than or equal to the $(i+1)$-st breakpoint go into the i-th category. Any values below the first breakpoint or above the last one are coded `NA` (a special R code for missing values). If the labels are omitted, R generates a suitable default of the form "(a, b]". By default the intervals are closed on the right, so our intervals are ≤ 4, 5-14, and 15+. To change this behavior, use the option `right = FALSE`.

Note that by specifying `fpe$effortg` on the left-hand-side, we have effectively added a new column to the `fpe` data frame.

Try fitting the analysis of covariance model:

```
> covfit <- lm( change ~ setting + effortg, data = fpe)
> covfit


Call:
lm(formula = change ~ setting + effortg, data = fpe)

Coefficients:
    (Intercept)          setting  effortgmoderate    effortgstrong
        -5.9540           0.1693           4.1439          19.4476
```

As you can see, `effortg` has been treated automatically as a factor, and R has generated the necessary dummy variables for "moderate" and "strong" programs, treating "weak" as the reference cell.

Choice of Contrasts: R codes unordered factors using the reference cell or "treatment contrast" method. The reference cell is always the first category which, depending on how the factor was created, is usually the first in alphabetical order. If you don't like this choice, R provides a special function to re-order levels, check out `help(relevel)`.

You can obtain a hierarchical anova table for the analysis of covariance model using the `anova()` function:

```
> anova(covfit)

Analysis of Variance Table

Response: change
```

```
          Df  Sum Sq Mean Sq F value    Pr(>F)
setting    1 1201.08 1201.08  36.556 1.698e-05 ***
effortg    2  923.43  461.71  14.053 0.0002999 ***
Residuals 16  525.69   32.86
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Type `?anova` to learn more about this function.

4.5 Regression Splines

The real power of R begins to shine when you consider some of the other functions you can include in a model formula. For starters, you can include mathematical functions, for example `log(setting)` is a perfectly legal term in a model formula. You don't have to create a variable representing the log of setting and then use it, R will create it 'on the fly', so you can type

```
> lm( change ~ log(setting) + effort, data = fpe)


Call:
lm(formula = change ~ log(setting) + effort, data = fpe)

Coefficients:
 (Intercept)  log(setting)        effort
     -61.737        15.638         1.002
```

If you wanted to use orthogonal polynomials of degree 3 on setting, you could include a term of the form `poly(setting, 3)`.

You can also get R to calculate a well-conditioned basis for regression splines. First you must load the `splines` library.

```
> library(splines)
```

This makes available the function `bs()` to generate B-splines. For example the call

```
> fpe$setting.bs <- bs(fpe$setting, knots = c(66, 74, 84))
```

will generate cubic B-splines with interior knots placed at 66, 74 and 84. This basis will use seven degrees of freedom, four corresponding to the constant, linear, quadratic and cubic terms, plus one for each interior knot. Alternatively, you may specify the number of degrees of freedom you are willing to spend on the fit using the parameter `df`. For cubic splines R will choose df-4 interior knots placed at suitable quantiles. You can also control the degree of the spline using the parameter `degree`, the default being cubic.

If you like natural cubic splines, you can obtain a well-conditioned basis using the function `ns()`, which has exactly the same arguments as `bs()` except for `degree`, which is always three. To generate a natural spline with five degrees of freedom, use the call

```
> fpe$setting.ns <- ns(fpe$setting, df=5)
```

Natural cubic splines are better behaved than ordinary splines at the extremes of the range. The restrictions mean that you save four degrees of freedom. You will probably want to use two of them to place additional knots at the extremes, but you can still save the other two.

To fit an additive model to fertility change using natural cubic splines on setting and effort with only one interior knot each, placed exactly at the median of each variable, try the following call:

```
> splinefit <- lm( change ~ ns(setting, knot=median(setting)) +
+   ns(effort, knot=median(effort)), data = fpe )
```

Here we used the parameter `knot` to specify where we wanted the knot placed, and the function `median()` to calculate the median of setting and effort. All calculations are done "on the fly".

Do you think the spline model is a good fit? Natural cubic splines with exactly one interior knot require the same number of parameters as an ordinary cubic polynomial, but are much better behaved at the extremes.

4.6 Other Options

The `lm()` function has several additional parameters that we have not discussed. These include

- `subset` to restrict the analysis to a subset of the data
- `weights` to do weighted least squares

and many others; see `help(lm)` for further details. The `args()` function lists the arguments used by any function, in case you forget them. Try `args(lm)`.

The fact that R has powerful matrix manipulation routines means that one can do many of these calculations from first principles. The next couple of lines create a model matrix to represent the constant, setting and effort, and then calculate the OLS estimate of the coefficients as $(X'X)^{-1}X'y$:

```
> X <- cbind(1, fpe$effort, fpe$setting)
> solve( t(X) %*% X ) %*% t(X) %*% fpe$change
```

```
            [,1]
[1,] -14.4510978
[2,]   0.9677137
[3,]   0.2705885
```

Compare these results with `coef(lmfit)`.

5 Generalized Linear Models

Generalized linear models are just as easy to fit in R as ordinary linear model. In fact, they require only an additional parameter to specify the variance and link functions.

5.1 Variance and Link Families

The basic tool for fitting generalized linear models is the `glm()` function, which has the folllowing general structure:

```
> glm(formula, family, data, weights, subset, ...)
```

where `...` stands for more esoteric options. The only parameter that we have not encountered before is `family`, which is a simple way of specifying a choice of variance and link functions. There are six choices of family:

Family	Variance	Link
gaussian	gaussian	identity
binomial	binomial	logit, probit or cloglog
poisson	poisson	log, identity or sqrt
gamma	gamma	inverse, identity or log
inverse.gaussian	inverse.gaussian	$1/\mu^2$
quasi	user-defined	user-defined

As can be seen, each of the first five choices has an associated variance function (for binomial, the binomial variance $\mu(1-\mu)$, and one or more choices of link functions (for binomial, the logit, probit or complementary log-log links).

As long as you want the default link, all you have to specify is the family name. If you want an alternative link, you must add a link argument. For example to do probits you use

```
> glm(formula, family = binomial(link = probit))
```

The last family on the list, `quasi`, is there to allow fitting user-defined models by maximum quasi-likelihood.

5.2 Logistic Regression

We will illustrate fitting logistic regression models using the contraceptive use data excerpted below (and shown in full further below):

```
  age education wantsMore notUsing using
  <25       low       yes       53     6
  <25       low        no       10     4
  ... lines omitted ...
40-49      high        no       12    31
```

The data are available on my website in a file called `cuse.dat`, and can be read directly from within R:

```
> cuse <- read.table("https://grodri.github.io/datasets/cuse.dat", header
+ = TRUE)
> cuse
```

```
      age education wantsMore notUsing using
1    <25       low       yes       53     6
2    <25       low        no       10     4
3    <25      high       yes      212    52
4    <25      high        no       50    10
5  25-29       low       yes       60    14
6  25-29       low        no       19    10
7  25-29      high       yes      155    54
8  25-29      high        no       65    27
9  30-39       low       yes      112    33
10 30-39       low        no       77    80
11 30-39      high       yes      118    46
12 30-39      high        no       68    78
13 40-49       low       yes       35     6
14 40-49       low        no       46    48
15 40-49      high       yes        8     8
16 40-49      high        no       12    31
```

I specified the `header` parameter as `TRUE`, because otherwise it would not have been obvious that the first line in the file has the variable names. There are no row names specified, so the rows will be numbered from 1 to 16. I also printed the data to make sure we got it alright.

Strings as Factors. We encountered *factors*, or categorical variables that take one of a discrete number of *levels*, in Section 4.4. Internally a factor is represented as an integer vector with the levels as an attribute. Versions of R prior to 4 would automatically read all strings as factors, but the default in functions such as `read.table()` is now `StringAsFactor=FALSE`. Because we kept the default, variables such as `education` have been read as strings, with actual values `low` and `high`. Modeling functions treat strings pretty much the same as factors, but if necessary one can convert a string variable to a factor using the `as.factor()` function.

Let us try a simple additive model where contraceptive use depends on age, education and whether or not the woman wants more children:

```
> lrfit <- glm( cbind(using, notUsing) ~ age + education + wantsMore,
+   data = cuse, family = binomial)
```

There are a few things to explain here. First, the function is `glm()` and I have assigned its value to an object called `lrfit` (for logistic regression fit). The first argument of the function is a model formula, which defines the response and linear predictor.

With binomial data the response can be either a vector or a matrix with two columns.

- If the response is a *vector*, it can be numeric with 0 for failure and 1 for success, or a factor with the first level representing "failure" and all others representing "success". In these cases R generates a vector of ones to represent the binomial denominators.
- Alternatively, the response can be a *matrix* where the first column is the number of "successes" and the second column is the number of "failures". In this case R adds the two columns together to produce the correct binomial denominator.

Because the latter approach is clearly the right one for us, I used the function `cbind()` to create a matrix by binding the column vectors containing the numbers using and not using contraception.

Following the special symbol `~` that separates the response from the predictors, we have a standard Wilkinson-Rogers model formula. In this case we are specifying main effects of `age`, `education` and `wantsMore`. Because all three predictors are string vectors they are treated automatically as categorical variables and represented using indicators for the categories, as you can see by inspecting the results:

```
> lrfit


Call:  glm(formula = cbind(using, notUsing) ~ age + education + wantsMore,
    family = binomial, data = cuse)

Coefficients:
 (Intercept)      age25-29      age30-39      age40-49  educationlow
     -0.8082        0.3894        0.9086        1.1892       -0.3250
wantsMoreyes
     -0.8330

Degrees of Freedom: 15 Total (i.e. Null);  10 Residual
Null Deviance:      165.8
Residual Deviance: 29.92    AIC: 113.4
```

R sorts the levels of a factor or string variable in alphabetical order, Because `<25` comes before `25-29`, `30-39`, and `40-49`, it has been picked as the reference cell for `age`. Similarly, `high` is the reference cell for `education` because high comes alphabetically before `low`! Finally, R picked `no` as the base for `wantsMore`.

If you are unhappy about these choices, which are admittedly not ideal, you can

(1) convert the variable to a factor and then change the reference cell using `relevel()`; for example for education we could set "low" as the reference by coding `cuse$education <- relevel(as.factor(cuse$education), "low")`, or

(2) define your own indicator variables.

I will use the second approach, defining indicators for women with high education, and for and women who want no more children, both added to the `cuse` data frame:

```
> cuse$noMore <- cuse$wantsMore == "no"
> cuse$hiEduc <- cuse$education == "high"
```

Now try the model with these predictors

```
> glm(cbind(using, notUsing) ~ age + hiEduc + noMore,
+   data = cuse, family = binomial)


Call:  glm(formula = cbind(using, notUsing) ~ age + hiEduc + noMore,
    family = binomial, data = cuse)

Coefficients:
(Intercept)     age25-29     age30-39     age40-49   hiEducTRUE   noMoreTRUE
    -1.9662       0.3894       0.9086       1.1892       0.3250       0.8330

Degrees of Freedom: 15 Total (i.e. Null);  10 Residual
Null Deviance:      165.8
Residual Deviance: 29.92    AIC: 113.4
```

Our indicator for high education is a Boolean variable that takes the values `FALSE` and `TRUE`. The corresponding coefficient is labeled `hiEducTRUE` to make it clear that it represents the case when the condition is true. (Alternatively, we could make the indicator take the values `0` and `1` by using `as.numeric()`, coding for example `cuse$hiEduc <- as.numeric(cuse$education == "high")`. In this case the coefficient would be labeled just `hiEduc`.)

The residual deviance of 29.92 on 10 d.f. is highly significant, so the additive model does not fit the data.

```
> pchisq(29.92, 10, lower.tail = FALSE)

[1] 0.0008828339
```

To obtain a p-value I specified `lower.tail` as `FALSE`. This is more accurate than computing the default lower tail and subtracting from one.

So, we need a better model. One of my favorites for this dataset introduces an interaction between age and wanting no more children, which is easily specified:

```
> lrfit2 <- glm( cbind(using, notUsing) ~ age * noMore + hiEduc , data = cuse,
+   family = binomial)
> lrfit2


Call:  glm(formula = cbind(using, notUsing) ~ age * noMore + hiEduc,
    family = binomial, data = cuse)

Coefficients:
```

```
        (Intercept)             age25-29             age30-39
           -1.80317              0.39460              0.54666
           age40-49           noMoreTRUE           hiEducTRUE
            0.57952              0.06622              0.34065
age25-29:noMoreTRUE  age30-39:noMoreTRUE  age40-49:noMoreTRUE
            0.25918              1.11266              1.36167

Degrees of Freedom: 15 Total (i.e. Null);  7 Residual
Null Deviance:      165.8
Residual Deviance: 12.63    AIC: 102.1
```

Note how R built the interaction terms automatically, and even came up with sensible labels for them. The model's deviance of 12.63 on 7 d.f. is not significant at the conventional five per cent level, so we have no evidence against this model.

To obtain more detailed information about this fit, try the `summary()` function:

```
> summary(lrfit2)
```

```
Call:
glm(formula = cbind(using, notUsing) ~ age * noMore + hiEduc,
    family = binomial, data = cuse)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-1.30027  -0.66163  -0.03286   0.81945   1.73851

Coefficients:
                   Estimate Std. Error z value Pr(>|z|)
(Intercept)        -1.80317    0.18018 -10.008  < 2e-16 ***
age25-29            0.39460    0.20145   1.959  0.05013 .
age30-39            0.54666    0.19842   2.755  0.00587 **
age40-49            0.57952    0.34742   1.668  0.09530 .
noMoreTRUE          0.06622    0.33071   0.200  0.84130
hiEducTRUE          0.34065    0.12577   2.709  0.00676 **
age25-29:noMoreTRUE 0.25918    0.40975   0.633  0.52704
age30-39:noMoreTRUE 1.11266    0.37404   2.975  0.00293 **
age40-49:noMoreTRUE 1.36167    0.48433   2.811  0.00493 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 165.77  on 15  degrees of freedom
Residual deviance:  12.63  on  7  degrees of freedom
AIC: 102.14
```

```
Number of Fisher Scoring iterations: 4
```

R follows the popular custom of flagging significant coefficients with one, two or three stars depending on their p-values. Try `plot(lrfit2)`. You get the same plots as in a linear model, but adapted to a generalized linear model; for example the residuals plotted are deviance residuals (the square root of the contribution of an observation to the deviance, with the same sign as the raw residual).

The functions that can be used to extract results from the fit include

- `residuals()` or `resid()`, for the deviance residuals
- `fitted()` or `fitted.values()`, for the fitted values (estimated probabilities)
- `predict()`, for the linear predictor (estimated logits)
- `coef()` or `coefficients()`, for the coefficients, and
- `deviance()`, for the deviance.

Some of these functions have optional arguments; for example, you can extract five different types of residuals, called "deviance", "pearson", "response" (defined as response - fitted value), "working" (the working dependent variable in the IRLS algorithm - linear predictor), and "partial" (a matrix of working residuals formed by omitting each term in the model). You specify the one you want using the `type` argument, for example `residuals(lrfit2, type = "pearson")`.

5.3 Model Updating

If you want to modify a model you may consider using the special function `update()`. For example to drop the `age:noMore` interaction in our model, one could use

```
> lrfit1 <- update(lrfit2, ~ . - age:noMore)
```

The first argument is the result of a fit, and the second an updating formula. The tilde `~` separates the response from the predictors, and the dot `.` refers to the right-hand side of the original formula, so here we simply remove `age:noMore`. Alternatively, one can give a new formula as the second argument.

The update function may also be used to fit the same model to different datasets, using the argument `data` to specify a new data frame. Another useful argument is `subset`, to fit the model to a different subsample. This function works with linear models as well as generalized linear models.

If you plan to fit a sequence of models you will find the anova function useful. Given a series of nested models, it will calculate the change in deviance between them. Try

```
> anova(lrfit1, lrfit2)

Analysis of Deviance Table

Model 1: cbind(using, notUsing) ~ age + noMore + hiEduc
Model 2: cbind(using, notUsing) ~ age * noMore + hiEduc
```

```
  Resid. Df Resid. Dev Df Deviance
1        10     29.917
2         7     12.630  3   17.288
```

Adding the interaction has reduced the deviance by 17.288 at the expense of 3 d.f.

If the argument to `anova()` is a single model, the function will show the change in deviance obtained by adding each of the terms in the order listed in the model formula, just as it did for linear models. Because this requires fitting as many models as there are terms in the formula, the function may take a while to complete its calculations.

The `anova()` function lets you specify an optional test. The usual choices will be "F" for linear models and "Chisq" for generalized linear models. Adding the parameter `test = "Chisq"` adds p-values next to the deviances. In our case

```
> anova(lrfit2, test = "Chisq")

Analysis of Deviance Table

Model: binomial, link: logit

Response: cbind(using, notUsing)

Terms added sequentially (first to last)

           Df Deviance Resid. Df Resid. Dev  Pr(>Chi)
NULL                         15    165.772
age         3   79.192        12     86.581 < 2.2e-16 ***
noMore      1   49.693        11     36.888 1.798e-12 ***
hiEduc      1    6.971        10     29.917 0.0082860 **
age:noMore  3   17.288         7     12.630 0.0006167 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We can see that all terms were highly significant when they were introduced into the model.

5.4 Model Selection

A very powerful tool in R is a function for stepwise regression that has three remarkable features:

- It works with generalized linear models, so it will do stepwise logistic regression, or stepwise Poisson regression,
- It understands hierarchical models, so it will only consider adding interactions after including the corresponding main effects in the models, and
- It understands terms involving more than one degree of freedom, so it it will keep together dummy variables representing the effects of a factor.

The basic idea of the procedure is to start from a given model (which could well be the null model) and take a series of steps, by either deleting a term already in the model, or adding a term from a list of candidates for inclusion, called the scope of the search and defined, of course, by a model formula.

Selection of terms for deletion or inclusion is based on Akaike's information criterion (AIC). R defines AIC as

AIC = –2 maximized log-likelihood + 2 number of parameters

The procedure stops when the AIC criterion cannot be improved.

In R all of this work is done by calling a couple of functions, `add1()` and `drop1()`, that consider adding or dropping one term from a model. These functions can be very useful in model selection, and both of them accept a `test` argument just like `anova()`.

Consider first `drop1()`. For our logistic regression model,

```
> drop1(lrfit2, test = "Chisq")

Single term deletions

Model:
cbind(using, notUsing) ~ age * noMore + hiEduc
           Df Deviance    AIC     LRT  Pr(>Chi)
<none>          12.630 102.14
hiEduc      1   20.099 107.61  7.4695 0.0062755 **
age:noMore  3   29.917 113.42 17.2877 0.0006167 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Obviously we can't drop any of these terms. Note that R considered dropping the main effect of education, and the age by want no more interaction, but did not examine the main effects of age or want no more, because one would not drop these main effects while retaining the interaction.

The sister function `add1()` requires a scope to define the additional terms to be considered. In our example we will consider all possible two-factor interactions:

```
> add1(lrfit2, ~ .^2, test = "Chisq")

Single term additions

Model:
cbind(using, notUsing) ~ age * noMore + hiEduc
             Df Deviance    AIC    LRT Pr(>Chi)
<none>            12.6296 102.14
age:hiEduc    3   5.7983 101.31 6.8313  0.07747 .
noMore:hiEduc 1  10.8240 102.33 1.8055  0.17905
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We see that neither of the missing two-factor interactions is significant by itself at the conventional five percent level. (However, they happen to be jointly significant.) Note that the model with the age by education interaction has a lower AIC than our starting model.

The `step()` function will do an automatic search. Here we let it start from the additive model and search in a scope defined by all two-factor interactions.

```
> search <- step(lrfit1, ~.^2)
```

The `step()` function produces detailed trace output that I have supressed. The returned object, however, includes an `anova` component that summarizes the search:

```
> search$anova
```

```
              Step Df  Deviance Resid. Df Resid. Dev      AIC
1                 NA        NA        10  29.917222 113.4251
2    + age:noMore -3 17.287669         7  12.629553 102.1375
3    + age:hiEduc -3  6.831288         4   5.798265 101.3062
4 + noMore:hiEduc -1  3.356777         3   2.441488  99.9494
```

As you can see, the automated procedure introduced, one by one, all three remaining two-factor interactions, to yield a final AIC of 99.9. This is an example where AIC, by requiring a deviance improvement of only 2 per parameter, may have led to overfitting the data.

Some analysts prefer a higher penalty per parameter. In particular, using `log(n)` instead of 2 as a multiplier yields BIC, the Bayesian Information Criterion. In our example `log(1607) = 7.38`, so we would require a deviance reduction of 7.38 per additional parameter. The `step()` function accepts `k` as an argument, with default 2. You may verify that specifying `k = log(1607)` leads to a much simpler model; not only are no new interactions introduced, but the main effect of education is dropped (even though it is significant).

In this example AIC would lead to a model that may be too complex, and BIC would lead to a model that may be too simple. In my opinion, the model with only one interaction is just right.

6 Next Steps

These notes have hardly scratched the surface of R, which has many more statistical functions. These include functions to calculate the density, cdf, and inverse cdf of distributions such as chi-squared, t, F, lognormal, logistic and others.

The `survival` library includes methods for the estimation of survival curves, tests of differences between survival curves, and Cox proportional hazards models. The library `lme4` includes code for fitting generalized linear mixed effect models, including multilevel models. Many new statistical procedures are first made available to the research community in the form of R functions.

To produce really nice graphs consider installing the `ggplot2` package. To draw a plot you specify a data frame, aesthetics that map variables to aspects of the graph, and geometries that specify whether to use points, lines, or other primitives. You fill find more information at https://ggplot2.tidyverse.org/

For data management I recommend that you install the `dplyr` package, which includes tools for adding new variables, selecting cases or variables (rows or columns), as well as summarizing and re-arranging your data. Check the overview at https://dplyr.tidyverse.org/.

You can also run `install.packages("tidyverse")` to install all the packages in the tidyverse, including `ggplot2` and `dplyr`, as well as `tidyr` (for help tidying data), `readr` (for reading rectangular data like csv files), `purrr` (for an alternative to loops), `tibble` (for tidy data frames), `stringr` (for working with strings) and `forcats` (for working with factors). Learn more at https://www.tidyverse.org/packages/.

In addition, R is a full-fledged programming language, with a rich complement of mathematical functions, matrix operations and control structures. It is very easy to write your own functions. To learn more about programming R, I recommend Wickham (2019)'s *Advanced R* book.

R is an interpreted language but it is reasonably fast, particularly if you take advantage of the fact that operations are vectorized, and try to avoid looping. Where efficiency is crucial you can always write a function in a compiled language such as C++ and then call it from R. Some of my work on multilevel generalized linear models used this approach.

Last, but most certainly not least, you will want to learn about dynamic documents using R Markdown. The basic idea here is to combine a narrative written in Markdown with R code, an approach that has excellent support in R Studio. The definite book on the subject is Xie, Allaire, and Grolemund (2019).

This tutorial has been written in R Markdown. You can download the source code `introducingR.Rmd`, the bibliography file `introducingR.bib`, and the image file `RStudioIDE.png` from GitHub. To reproduce the PDF document you also need `tweaks.tex` from the same source. To generate an HTML document instead, change the output specification near the top of the script.

References

Becker, Richard A., and John M. Chambers. 1984. *S an Interactive Environment for Data Analysis and Graphics*. Belmont, CA: Wadsworth.

Becker, Richard A., John M. Chambers, and Allan R. Wilks. 1988. *The New S Language*. Pacific Grove, CA: Wadsworth.

Braun, W. John, and Duncan J. Murdoch. 2016. *A First Course in Statistical Programming with R*. Second Edition. Cambridge University Press.

Chambers, John M. 1998. *Programming with Data*. New York: Springer.

———. 2008. *Software for Data Analysis. Programming with R*. New York: Springer.

———. 2016. *Extending R*. Boca Raton, FL: Chapman Hall/CRC.

Chambers, John M., and Trevor J. Hastie, eds. 1992. *Statistical Models in S*. Pacific Grove, CA: Wadsworth.

Dalgaard, Peter. 2008. *Introductory Statistics with R*. Second Edition. New York: Springer.

Fox, John. 2002. *An R and S-Plus Companion to Applied Regression*. Thousand Oaks, CA: SAGE.

Hothorn, Torsten, and Brian S. Everitt. 2014. *A Handbook of Statistical Analyses Using R*. Third Edition. Boca Raton, FL: CRC Press.

Krause, Anreas, and Melvin Olson. 1997. *The Basics of S and S-Plus*. New York: Springer.

Murrell, Paul. 2006. *R Graphics*. Boca Raton, FL: Chapman Hall/CRC.

Pinheiro, José C., and Douglas M. Bates. 2000. *Mixed-Effects Models in S and S-Plus*. New York: Springer.

Therneau, Terry M., and Patricia M. Grambsch. 2000. *Modeling Survival Data*. New York.

Venables, W. N., and B. D. Ripley. 2000. *S Programming*. New York: Springer.

———. 2002. *Modern Applied Statistics with S-Plus*. Fourth Edition. New York: Springer.

Wickham, Hadley. 2016. *ggplot2 Elegant Graphics for Data Analysis*. Second Edition. New York: Springer. https://ggplot2-book.org/.

———. 2019. *Advanced R*. Second Edition. Boca Raton, FL: CRC Press. https://adv-r.hadley.nz/.

Wickham, Hadley, and Garret Grolemund. 2017. *R for Data Science*. Sebastopol, CA: O'Reilly. https://r4ds.had.co.nz/.

Wilkinson, Leland. 2005. *The Grammar of Graphics*. Second Edition. New York: Springer.

Xie, Yihui, J. J. Allaire, and Garrett Grolemund. 2019. *R Markdown: The Definitive Guide*. Boca Raton, FL: Chapman Hall/CRC. https://bookdown.org/yihui/rmarkdown/.